Lecture 1

Useful Standard Python Libraries

Python has a number of built-in libraries, or collections of modules, that provide a wide range of functionality. Some of the most commonly used built-in libraries include:

- os: provides a way to interact with the operating system
- sys: provides access to system-specific parameters and functions
- math: provides mathematical functions and constants
- random: generates random numbers
- string: provides string-related functions
- re: provides regular expression matching operations
- json: provides encoding and decoding of JSON data
- datetime: provides classes for manipulating dates and times
- collections: provides alternatives to built-in types that can be more efficient in certain situations

Let's take a look at the Python Standard Library, which is a set of modules included with every Python installation. These modules provide a wide range of functionality and are designed to improve efficiency and ease of use. You learned about the first modules in Python classes, so now we will look at other modules.

collections: This library provides alternatives to built-in types that can be more efficient in certain situations. Some examples of classes in this library include Counter, which is a dict subclass for counting hashable objects, and defaultdict, which is a dict subclass that calls a factory function to supply missing values.

Counter: This class is a dict subclass for counting hashable objects. It is a container that will hold the count of each of the elements present in the container.

Example:

```
from collections import Counter
c = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    c[word] += 1
print(c)
# Output: Counter({'blue': 3, 'red': 2, 'green': 1})
```

defaultdict: this class is a dict subclass that calls a factory function to supply missing values. It is similar to the standard dictionary, but it has a default value for any new keys that are accessed.

Example:

```
from collections import defaultdict
d = defaultdict(int)
d['a'] = 1
d['b'] = 2
print(d['c']) # 0
```

itertools: this module provides a variety of functions for working with iterators, including functions for creating iterators, combining them, and working with infinite iterators. For example, itertools.count() returns an iterator that generates an infinite sequence of integers, starting with a given value, and itertools.product() returns the cartesian product of input iterables as an iterator.

cycle: this function from itertools returns an iterator that will cycle through the elements of an iterable indefinitely.

Example:

```
from itertools import cycle
colors = cycle(['red', 'blue', 'green'])
for i in range(7):
    print(next(colors))
# Output: red blue green red blue green red
```

repeat: this function from itertools returns an iterator that will repeat the same element a specified number of times or indefinitely if no number is specified

Example:

```
from itertools import repeat
for i in repeat('over', 5):
    print(i)
# Output: over over over over over
```

All these libraries are very useful and can be used in various situations. They provide functionality that is often needed in programming, and they are well-tested and optimized to be efficient.

re: The re module provides regular expression matching operations. Regular expressions are a powerful tool for matching patterns in text, and the re module provides a wide range of functions for working with regular expressions in Python. Some common functions in this module include search(), findall(), sub(), and compile().

Example:

python

Example:

```
import re

# Search for the first occurrence of a pattern
result = re.search(r'\d{3}-\d{2}-\d{4}', 'Social Security Number: 123-45-6789')
print(result.group(0))  # 123-45-6789

# Find all occurrences of a pattern
result = re.findall(r'\b\w+\b', 'This is a sentence with a few words')
print(result)  # ['This', 'is', 'a', 'sentence', 'with', 'a', 'few', 'words']

# Replace all occurrences of a pattern with a new string
result = re.sub(r'\d{3}-\d{2}-\d{4}', 'xxx-xx-xxxx', 'Social Security Number: 123-45-6789')
print(result)  # Social Security Number: xxx-xx-xxxx
```

json: The json module provides encoding and decoding of JSON (JavaScript Object Notation) data. JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. The json module provides functions for working with JSON data in Python, including load(), loads(), dump(), and dumps().

Example:

```
import json

# Serialize a Python object to a JSON formatted string
data = {'name': 'John', 'age': 30, 'city': 'New York'}
json_data = json.dumps(data)
print(json_data)  # {"name": "John", "age": 30, "city": "New York"}

# Deserialize a JSON string to a Python object
json_data = '{"name": "John", "age": 30, "city": "New York"}'
data = json.loads(json_data)
print(data)  # {'name': 'John', 'age': 30, 'city': 'New York'}
```

string: The string module provides string-related functions. It contains a number of useful string constants, such as ascii_letters, digits, and punctuation, as well as functions for working with strings, such as capwords(), join(), replace(), and split().

Example:

```
import string

# Capitalize the first letter of each word in a string
result = string.capwords('this is a test')
print(result)  # This Is A Test

# Concatenate a list of strings with a separator
words = ['hello', 'world']
```

All these libraries are very useful and can be used in various situations. They provide functionality that is often needed in programming, and they are well-tested and optimized to be efficient.

JSON and CSV

JSON (JavaScript Object Notation) and CSV (Comma-Separated Values) are two commonly used data formats for exchanging and storing data.

JSON is a human-readable, text-based format that is commonly used for data exchange in APIs and web applications. It uses key-value pairs to represent data and is often used for storing complex data structures, such as nested arrays and dictionaries. JSON is also easy to parse and generate using many programming languages, including Python.

CSV, on the other hand, is a simpler and more widely supported format. It stores data in a tabular form, where each line is a record and each field within a record is separated by a comma. This format is widely used in data analysis and spreadsheet applications, and is supported by most database and spreadsheet programs. The simplicity of CSV makes it easy to use and transfer data, but it is not well suited for representing complex data structures.

In summary, JSON is a more versatile and flexible data format, while CSV is simpler and more widely supported. The choice between JSON and CSV depends on the specific needs of the data being stored and the applications that will be accessing it.

Examples of when you might use JSON or CSV:

- **JSON: When you want to exchange data between a web application and a server**, you might use JSON. For example, you might use an API to fetch data from a server and receive the data in a JSON format. You can then use the JSON data in your web application.
- **CSV: When you want to analyze data in a spreadsheet program**, you might use CSV. For example, you might export data from a database and save it as a CSV file, which you can then open in a spreadsheet program like Microsoft Excel or Google Sheets. You can easily sort, filter, and visualize the data in the spreadsheet program.
- **JSON: When you want to store complex data structures**, you might use JSON. For example, you might use JSON to store information about a collection of books, where each book has a title, author, publication date, and list of genres. You can easily nest this data within the JSON format to represent the relationships between the different elements.

- **CSV: When you want to store data that is easy to process**, you might use CSV. For example, you might use CSV to store a list of names and addresses, where each line is a record and each field within a record is separated by a comma. You can easily manipulate this data using simple string operations and write it to a file for later use.

Examples comparing JSON and CSV formats in Python:
JSON:

```
import json

# A Python dictionary to be converted to JSON
person = {
  "first_name": "John",
  "last_name": "Doe",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA"
  }
}

# Convert the dictionary to JSON string
person_json = json.dumps(person)

# Load the JSON string to a Python dictionary
person_dict = json.loads(person_json)

print("Person JSON:", person_json)
print("Person Dict:", person_dict)
```

CSV:

```
import csv

# A list of dictionaries to be converted to CSV
people = [
  {
    "first_name": "John",
    "last_name": "Doe",
    "age": 30,
    "address": "123 Main St, Anytown, CA"
  },
  {
    "first_name": "Jane",
    "last_name": "Doe",
```

```
    "age": 28,
    "address": "456 Elm St, Anytown, CA"
  }
]

# Write the list of dictionaries to a CSV file
with open('people.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name', 'age', 'address']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    for person in people:
        writer.writerow(person)

# Read the CSV file to a list of dictionaries
with open('people.csv', 'r') as csvfile:
    reader = csv.DictReader(csvfile)
    people_list = list(reader)

print("People List:", people_list)
```

В приведенных выше примерах JSON используется для представления одного словаря, а CSV — для представления списка словарей. Преобразование между двумя форматами можно выполнить с помощью библиотек json и csv.

In the examples above, JSON is used to represent a single dictionary while CSV is used to represent a list of dictionaries. The conversion between the two formats can be done using the json and csv libraries.

Practical tasks on the topic of this lecture

Processing JSON data: Using the json module, write a script that reads in a JSON file containing information about books (title, author, and ISBN) and outputs a list of all the books written by a specific author.

Example:

```
import json

def find_books_by_author(file_path, author):
    with open(file_path, 'r') as f:
        data = json.load(f)
    books = []
```

```
    for book in data['books']:
        if book['author'] == author:
            books.append(book)
    return books

file_path = 'books.json'
author = 'J.K. Rowling'
books = find_books_by_author(file_path, author)
print(books)
```

Cleaning text data: Using the string module, write a script that takes in a text file and removes all punctuation and capitalization, and then write the cleaned text to a new file.
python
Example:

```
import string

def clean_text(file_path):
    with open(file_path, 'r') as f:
        text = f.read()
    text = text.lower()
    text = text.translate(text.maketrans('', '', string.punctuation))
    with open('cleaned_text.txt', 'w') as f:
        f.write(text)

file_path = 'text.txt'
clean_text(file_path)
```

Manipulating CSV data: Using the csv module, write a script that reads

Example:

```
import csv

def calculate_avg_income(file_path):
    with open(file_path, 'r') as f:
        reader = csv.DictReader(f)
        income_by_state = {}
        for row in reader:
            state = row['state']
            income = int(row['income'])
            if state in income_by_state:
                income_by_state[state]['total'] += income
                income_by_state[state]['count'] += 1
```

```
        else:
            income_by_state[state] = {'total': income, 'count': 1}
                avg_income_by_state = {state: income_by_state[state]['total'] / income_by_state[state]['count'] for state in income_by_state}
    return avg_income_by_state

file_path = 'income_data.csv'
avg_income = calculate_avg_income(file_path)
print(avg_income)
```

Extracting information from web pages: Using the re module, write a script that extracts all the phone numbers from a web page.

python

Example:

```
import re
import requests

def extract_phone_numbers(url):
    r = requests.get(url)
    phone_numbers = re.findall(r'\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}', r.text)
    return phone_numbers

url = 'https://example.com'
phone_numbers = extract_phone_numbers(url)
print(phone_numbers)
```

These are just a few examples of practical tasks that can be given using these modules. The specific tasks will depend on the skill level of the students and the goals of the course. It's also worth mentioning that these examples are simplified versions of what one might encounter in a real-world scenario.

Counting occurrences of items in a list: Using the Counter class, write a script that takes in a list of items and returns a dictionary of the items and their occurrences in the list.

Example:

```
from collections import Counter

def count_items(items):
    return Counter(items)

items = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1]
counts = count_items(items)
print(counts)
```

Accessing missing dictionary keys: Using the defaultdict class, write a script that takes in a list of items and a default value, and returns a dictionary of the items and their occurrences in the list. If an item is not in the dictionary, it should return the default value.

Example:

```
from collections import defaultdict

def count_items(items, default_value):
    counts = defaultdict(lambda: default_value)
    for item in items:
        counts[item] += 1
    return counts

items = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1]
default_value = 0
counts = count_items(items, default_value)
print(counts)
```

Iterating over combinations: Using the itertools module, write a script that takes in a list of items and a number n, and returns all possible combinations of n items from the list.

Example:

```
import itertools

def get_combinations(items, n):
    return list(itertools.combinations(items, n))

items = [1, 2, 3, 4]
n = 2
combinations = get_combinations(items, n)
print(combinations)
```

Repeating an iterator: Using the itertools module, write a script that takes in an iterator and a number n, and returns the iterator repeated n times.

Example:

```
import itertools

def repeat_iterator(iterable, n):
    return itertools.chain.from_iterable(itertools.repeat(iterable, n))

iterable = [1, 2, 3]
```

```
n = 2
repeated_iterable = repeat_iterator(iterable, n)
print(list(repeated_iterable))
```

Cycling through an iterator: Using the itertools module, write a script that takes in an iterator and returns an infinite iterator that cycles through the input iterator.
Example:

```
import itertools

def cycle_iterator(iterable):
    return itertools.cycle(iterable)

iterable = [1, 2, 3]
cycled_iterable = cycle_iterator(iterable)
print(next(cycled_iterable))
print(next(cycled_iterable))
print(next(cycled_iterable))
print(next(cycled_iterable))
```

Lecture 2

Efficient Data Structures

pandas and NumPy are two popular Python libraries that are widely used in data analysis and manipulation.

Pandas is a library for data manipulation and analysis. It provides data structures and functions for handling and manipulating numerical tables and time series data. It is built on top of NumPy and provides easy-to-use data structures and data analysis tools for handling and manipulating numerical tables and time series data.

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Grouping and aggregation: pandas provides powerful tools for grouping and aggregating data. You can group data by one or more columns and apply various aggregation functions to each group. For example, you can group a dataset by the values in a column and then calculate the mean of each group.

Example:

```
import pandas as pd
df = pd.read_csv('data.csv')
grouped_data = df.groupby('column_name').mean()
print(grouped_data)
```

Handling missing data: pandas provides various options for handling missing data, such as filling in missing values with a specific value or interpolating the missing values.

Example:

```
import pandas as pd
df = pd.read_csv('data.csv')
df = df.fillna(value=0)
print(df)
```

Merging and joining data: pandas provides various options for merging and joining data from multiple DataFrames or datasets. For example, you can join two DataFrames on a specific column, similar to a SQL JOIN.

Example:

```
import pandas as pd
df1 = pd.read_csv('data1.csv')
df2 = pd.read_csv('data2.csv')
```

```
merged_data = pd.merge(df1, df2, on='column_name')
print(merged_data)
```

Data transformation: pandas provides various functions for transforming data, such as pivoting and melting data. For example, you can pivot a DataFrame to reshape it into a different format.

Example:

```
import pandas as pd
df = pd.read_csv('data.csv')
pivoted_data = df.pivot(index='column_name1', columns='column_name2', values='column_name3')
print(pivoted_data)
```

pivot is a DataFrame method in the pandas library that can be used to reshape a DataFrame. It allows you to transform a DataFrame from a "long" format to a "wide" format or vice versa.

The basic syntax for using the pivot method is as follows:

```
pivot_table = df.pivot(index='index_column', columns='column_to_become_columns', values='column_to_become_values')
```

Here, index_column is the column that will become the new DataFrame's index, column_to_become_columns is the column that will become the new DataFrame's columns, and column_to_become_values is the column that will populate the new DataFrame's cells.

For example, let's say you have a DataFrame with three columns: 'date', 'product', and 'sales'. To pivot this DataFrame so that the 'date' column becomes the index, the 'product' column becomes the columns, and the 'sales' column becomes the values, you would use the following code:

```
import pandas as pd
df = pd.read_csv('data.csv')
pivoted_data = df.pivot(index='date', columns='product', values='sales')
print(pivoted_data)
```

Another useful feature of the pivot function is the ability to pass an optional argument aggfunc to it. This allows you to specify the aggregation function that is applied to duplicate values. By default, pivot will raise an error if there are duplicate values.

For example, if you have the following DataFrame:

Product	Sales	Date

0 Apple 100 1
1 Apple 200 1
2 Pear 50 2
3 Pear 30 2

You can use the following code to pivot this dataframe and sum the duplicate values:

```
import pandas as pd
df = pd.read_csv('data.csv')
pivoted_data = df.pivot_table(index='Date', columns='Product', values='Sales',aggfunc='sum')
print(pivoted_data)
```

The resulting pivot table would be:

```
Product  Apple  Pear
Date
1          300   NaN
2           NaN   80
```

Note that pivot_table is similar to pivot but will handle duplicate values by applying the specified aggregation function.

pivot and pivot_table are powerful tools for reshaping and aggregating data in pandas. These methods can be used to transform a DataFrame into a format that is more suitable for a specific task or analysis.

Data visualization: pandas integrates well with data visualization libraries such as matplotlib and seaborn, which allows you to easily create plots and charts from your DataFrames.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df.plot(kind='bar',x='column_name1',y='column_name2')
plt.show()
```

This is just a small sample of the capabilities of pandas. It's a very versatile library and can be used to perform a wide range of data manipulation and analysis tasks.

matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK.

It provides a range of tools for creating static, animated, and interactive visualizations in Python. Some of the most common types of plots that can be created with matplotlib include:

- Line plots: for displaying data that changes over time or other continuous variables
- Scatter plots: for displaying the relationship between two or more variables
- Bar plots: for displaying data that is divided into different categories
- Histograms: for displaying the distribution of a single variable
- Pie charts: for displaying the proportion of different categories in a dataset

Example of how to use matplotlib to create a simple line plot:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the data
ax.plot(x, y)

# Add labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Simple Line Plot')

# Show the plot
plt.show()
```

matplotlib also has an extensive customization options, you can easily change the color, line style, and marker style of your plots. You can also add gridlines, legends, and annotations to your plots. Also, you can use a variety of pre-defined styles and color palettes, such as "ggplot" or "seaborn".

There are other library like seaborn that built on top of matplotlib, it provides a high-level interface for drawing attractive and informative statistical graphics.

For more advanced usage, matplotlib also has capabilities for creating 3D plots, subplots, and other advanced visualizations. With a little bit of creativity and knowledge of the library's capabilities, you can create all sorts of interesting and informative visualizations with matplotlib.

For practical tasks, you can use matplotlib to visualize data from a CSV file, visualize the distribution of a dataset, or compare multiple datasets in a single plot. You can also use it to create plots of mathematical functions or simulate simple animations.

Visualizing data from a CSV file:

```
import matplotlib.pyplot as plt
import pandas as pd

# Read data from a CSV file
data = pd.read_csv('data.csv')

# Create a scatter plot of the data
plt.scatter(data['x'], data['y'])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot of Data')
plt.show()
```

Visualizing the distribution of a dataset:

```
import matplotlib.pyplot as plt
import numpy as np

# Create a random dataset
data = np.random.normal(50, 10, 100)

# Create a histogram of the data
plt.hist(data, bins=20)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Data')
plt.show()
```

Comparing multiple datasets in a single plot:

```
import matplotlib.pyplot as plt
import numpy as np

# Create two random datasets
data1 = np.random.normal(50, 10, 100)
data2 = np.random.normal(60, 15, 100)
```

```
# Create a figure and axis
fig, ax = plt.subplots()

# Plot both datasets on the same axis
ax.plot(data1, label='Dataset 1')
ax.plot(data2, label='Dataset 2')

# Add labels, title, and legend
ax.set_xlabel('Index')
ax.set_ylabel('Value')
ax.set_title('Comparison of Two Datasets')
ax.legend()

# Show the plot
plt.show()
```

Plotting a mathematical function:

```
import matplotlib.pyplot as plt
import numpy as np

# Create x and y data
x = np.linspace(-np.pi, np.pi, 100)
y = np.sin(x)

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the data
ax.plot(x, y)

# Add labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Plot of the Sine Function')

# Show the plot
plt.show()
```

Simulating a simple animation:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Create x and y data
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

# Create a figure and axis
fig, ax = plt.subplots()

# Initialize a line object
line, = ax.plot(x, y)

# Define an animation function
def animate(i):
    line.set_ydata(np.sin(x + i / 10))
    return line,

# Create an animation object
ani = animation.FuncAnimation(fig, animate, frames=100, blit=True)

# Show the animation
plt.show()
```

You can find more information and examples on the official website of matplotlib: https://matplotlib.org/stable/contents.html

Practical tasks on the topic of this lecture

Loading a CSV file into a DataFrame: Using pandas, write a script that loads a CSV file into a DataFrame and displays the first 5 rows of the DataFrame.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.head())
```

Selecting columns from a DataFrame: Using pandas, write a script that selects specific columns from a DataFrame and displays them.
python

Example:

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
columns_to_select = ['col1', 'col2', 'col3']
selected_columns = df[columns_to_select]
print(selected_columns)
```

Filtering rows from a DataFrame: Using pandas, write a script that filters rows from a DataFrame based on a specific condition and displays the resulting DataFrame.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')
condition = df['col1'] > 5
filtered_df = df[condition]
print(filtered_df)
```

Calculating statistics on a NumPy array: Using NumPy , write a script that loads a CSV file into a NumPy array, and calculates the mean, standard deviation, and maximum value of the array.

Example:

```
import numpy as np

data = np.genfromtxt('data.csv', delimiter=',')
mean = np.mean(data)
std = np.std(data)
max_value = np.max(data)
print("Mean:", mean)
print("Standard deviation:", std)
print("Max value:", max_value)
```

Matrix operations: Using NumPy, write a script that creates a matrix and performs basic mathematical operations such as addition, subtraction, multiplication and transpose of matrix.

```
import numpy as np

matrix1 = np.array([[1,2,3],[4,5,6]])
matrix2 = np.array([[7,8,9],[10,11,12]])

#Addition
add_matrix = matrix1 + matrix2
print("Addition of matrices:", add_matrix)

#Subtraction
```

```
sub_matrix = matrix1 - matrix2
print("Subtraction of matrices:", sub_matrix)

#multiplication
mul_matrix = matrix1 * matrix2
print("Multiplication of matrices:", mul_matrix)
```

Lecture 3

Working with MS Office objects

The **win32com** module in Python allows you to interact with Microsoft Office applications such as Word, Excel, and PowerPoint using the Component Object Model (COM) interface.

For example, you can use it to open an existing Word document, add text, change the font, and save the document:

```
import win32com.client as win32

word = win32.gencache.EnsureDispatch('Word.Application')
doc = word.Documents.Open('C:\\path\\to\\document.docx')
word.Visible = True

range = doc.Range()
range.InsertAfter("Hello, world!")
range.Font.Name = "Arial"
range.Font.Size = 14

doc.Save()
doc.Close()
word.Quit()
```

With Excel, you can open an existing workbook, add data to a worksheet, create charts and graphs, and save the workbook:

```
import win32com.client as win32

excel = win32.gencache.EnsureDispatch('Excel.Application')
workbook = excel.Workbooks.Open('C:\\path\\to\\workbook.xlsx')
excel.Visible = True

worksheet = workbook.Worksheets("Sheet1")
worksheet.Cells(1, 1).Value = "Name"
worksheet.Cells(1, 2).Value = "Age"
worksheet.Cells(2, 1).Value = "John"
worksheet.Cells(2, 2).Value = 30

chart = worksheet.ChartObjects().Add(0,0,300,300).Chart
chart.ChartType = win32.constants.xlColumnClustered
chart.SetSourceData(worksheet.Range("A1:B2"))

workbook.Save()
```

```
workbook.Close()
excel.Quit()
```

You can also use win32com to interact with PowerPoint, for example, you can create a new presentation, add slides, add text and images, and save the presentation:

```
import win32com.client as win32

ppt = win32.gencache.EnsureDispatch('PowerPoint.Application')
presentation = ppt.Presentations.Add()
ppt.Visible = True

slide = presentation.Slides.Add(1, win32.constants.ppLayoutTitleOnly)
title = slide.Shapes.Title
title.TextFrame.TextRange.Text = "My Presentation"

slide = presentation.Slides.Add(2, win32.constants.ppLayoutBlank)
textbox = slide.Shapes.AddTextbox(1,20,200,100,50)
textbox.TextFrame.TextRange.Text = "Hello, World!"

image = slide.Shapes.AddPicture("C:\\path\\to\\image.jpg", False, True, 20, 20, 100, 100)

presentation.SaveAs("C:\\path\\to\\presentation.pptx")
presentation.Close()
ppt.Quit()
```

Working with .docx and .xlsx files in Python can be done using the python-docx and openpyxl libraries, respectively.

The **python-docx** library allows you to read and edit existing .docx files, and also create new .docx files from scratch. You can add text, paragraphs, lists, tables, and images to the .docx file.

For example, you can use the following code to create a new .docx file and add a heading and some text to it:

```
from docx import Document

document = Document()
document.add_heading('My Heading', 0)
document.add_paragraph('My paragraph')
document.save('My document.docx')
```

The **openpyxl** library allows you to read and write .xlsx files. You can access cells, rows, and columns in the spreadsheet and modify their values. You can also create new spreadsheets, and add charts, formulas, and images to them.

For example, you can use the following code to read the data from an existing .xlsx file and print the value of the cell in the first row and first column:

```
from openpyxl import load_workbook

workbook = load_workbook('data.xlsx')
worksheet = workbook.active
cell_value = worksheet.cell(row=1, column=1).value
print(cell_value)
```

Aspose.Words is a .NET library that provides the ability to manipulate Microsoft Word documents. It offers a wide range of features and benefits, including the ability to work with styles in Word documents.

One of the key features of Aspose.Words is the ability to create, modify, and format styles in a Word document. You can create custom styles or modify existing ones to suit your needs. Additionally, you can apply styles to text, paragraphs, or tables in a document to maintain consistency and simplify formatting.

Another benefit of using Aspose.Words is the ability to perform operations on Word documents programmatically. For example, you can insert or delete text, add or modify images, and update tables and lists. You can also perform operations on document elements such as paragraphs, tables, and sections.

Of the key features of Aspose.Words is its ability to work with styles in Word documents. With Aspose.Words, you can easily access and modify the styles in a document, including the font, size, color, and alignment. You can also create new styles and apply them to specific portions of text in a document.

For example, to change the font of all instances of the "Heading 1" style in a document, you can use the following code:

```
// Load the document
Document doc = new Document("input.docx");

// Access the "Heading 1" style
Style heading1 = doc.getStyles().getByName("Heading 1");

// Change the font of the style
```

```
heading1.getFont().setName("Arial");

// Save the updated document
doc.save("output.docx");
```

In this example, the Document class is used to load the input document, and the getStyles method is used to access the styles defined in the document. The getByName method is then used to access the "Heading 1" style, and the getFont method is used to access the font associated with the style. Finally, the setName method is used to change the font name, and the save method is used to save the updated document.

Overall, Aspose.Words provides a comprehensive set of features for working with Word documents and styles, making it an ideal solution for developers who need to automate Word document processing tasks.

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects, which have properties (also known as attributes) and methods. In Python, classes are used to define objects and their properties and methods.

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

Here is an example of a simple class in Python:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print("Woof!")

dog1 = Dog("Fido", "Golden Retriever")
dog1.bark() # Output: Woof!
```

In this example, we define a class Dog with two attributes name and breed. The __init__ method is a special method that is called when a new object of this class is created. It initializes the attributes of the object. The class also has a method bark, which prints "Woof!" when called.

We can then create an object of the class Dog by calling the class name as a function, and passing in the required arguments for the __init__ method.

In OOP, classes inherit from other classes and can add or override their methods and attributes. This allows for a clean and organized code structure with less repetitive code. You can also use inheritance to create a class hierarchy, which organizes classes according to their functionality and relationships.

Example of a class for a bank account, with methods for deposit, withdraw and check balance:

```
class BankAccount:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} has been deposited to {self.name}'s account")

    def withdraw(self, amount):
        if amount > self.balance:
            print(f"Insufficient balance in {self.name}'s account")
        else:
            self.balance -= amount
            print(f"{amount} has been withdrawn from {self.name}'s account")

    def check_balance(self):
        print(f"{self.name}'s current balance is {self.balance}")

account1 = BankAccount("John Doe", 1000)
account1.check_balance() # Output: John Doe's current balance is 1000
account1.deposit(500) # Output: 500 has been deposited to John Doe's account
account1.check_balance() # Output: John Doe's current balance is 1500
account1.withdraw(2000) # Output: Insufficient balance in John Doe's account
account1.withdraw(500) # Output: 500 has been withdrawn from John Doe's account
account1.check_balance() # Output: John Doe's current balance is 1000
```

This class has three methods: deposit, withdraw, and check_balance. Each method performs a specific action on the bank account object. The deposit method takes an amount as input and adds it to the balance of the account. The withdraw method takes an amount as input, checks if the account has sufficient balance, and if so, subtracts the amount from the balance. The check_balance method simply prints the current balance of the account.

In this example we create an object account1 of class BankAccount and call the methods on the object to demonstrate how the class works.

Another example is creating a class for a rectangle, with attributes for length and width, and methods for calculating area and perimeter:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

rect1 = Rectangle(5, 10)
print(rect1.calculate_area()) # Output: 50
print(rect1.calculate_perimeter()) # Output: 30
```

In this example, the class Rectangle has two attributes: length and width. It also has two methods calculate_area and calculate_perimeter which return the area and perimeter of the rectangle respectively.

In object-oriented programming (OOP), the concepts of encapsulation, inheritance, and polymorphism play a fundamental role in creating efficient, maintainable, and scalable code.

Encapsulation: Encapsulation refers to the concept of bundling data and methods that work on that data within a single unit, or object. Here is an example of a simple class that implements encapsulation in Python:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self._speed = 0

    def accelerate(self):
        self._speed += 10

    def brake(self):
        self._speed -= 10

    def get_speed(self):
```

```
        return self._speed
```

Polymorphism refers to the ability of different objects to respond to the same method call in different ways. Here is an example of polymorphism in Python:

```
def show_shape_area(shape):
    print("Area:", shape.area())

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

rect = Rectangle(10, 20)
circle = Circle(5)

show_shape_area(rect)
show_shape_area(circle)
```

Inheritance refers to the ability to define a new class that is a modified version of an existing class. Here is an example of inheritance in Python:

```
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius
```

Decorators are a way to modify the behavior of a function or class. Here is an example of a simple decorator in Python:

```
def uppercase(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper

@uppercase
def greet():
    return "Hello, World!"

print(greet())
```

Operator overloading refers to the ability to define custom behavior for the built-in operators in Python. Here is an example of operator overloading in Python:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3)  # Output: Vector(4, 6)
```

In this example, we have defined a class Vector that represents a 2D vector. The __add__ method is used to define the behavior of the + operator when it is applied to two Vector objects. The __str__ method is used to define how a Vector object should be represented as a string, which is useful for debugging and testing purposes.

Magic Methods are special methods in Python that have a double underscore prefix and suffix, such as init, str, len, etc. They are used to define special behavior for built-in functions and operators. Here is an example of using the str magic method in Python:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} by {self.author}, {self.pages} pages"

book = Book("The Great Gatsby", "F. Scott Fitzgerald", 180)
print(book)
```

Exception Classes are a way to handle errors in a program. In Python, you can define your own custom exception classes by inheriting from the built-in Exception class. Here is an example of a custom exception class in Python:

```
class InvalidAgeException(Exception):
    pass

age = int(input("Enter your age: "))
if age < 0:
    raise InvalidAgeException("Age cannot be negative")
```

Context Managers are a way to manage resources, such as files, sockets, and databases, in a clean and efficient manner. In Python, you can define your own custom context managers by using the with statement and implementing the __enter__ and __exit__ magic methods. Here is an example of a custom context manager in Python:

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
```

```
    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, exc_tb):
        self.file.close()

with FileManager("example.txt", "w") as file:
    file.write("Hello, World!")
```

Enums are a way to define a set of named constants in Python. Enums are created using the enum module in Python 3.4 or later, or using the enum34 backport package for earlier versions of Python. Here is an example of using enums in Python:

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(Color.RED)
print(Color.RED.value)
```

Practical tasks for OOP:

Create a class for a bank account, with methods for deposit, withdraw and check balance.
Create a class for a car, with methods for starting, stopping, and honking.
Create a class for a person, with attributes for name, age, and address.
Create a class for a university, with attributes for name, address, and students.
Create a class for a rectangle, with attributes for length and width, and methods for calculating area and perimeter.
Create a class for a shopping cart, with methods for adding and removing items, and calculating the total price.
Create a class for a game, with attributes for score, lives, and level, and methods for starting, ending, and restarting the game.

Дополнительную информацию и примеры вы можете найти на официальном сайте python: https://docs.python.org/3/tutorial/classes.html.

Lecture 4

SciPy and scikit-learn are two popular Python libraries for optimization, linear algebra, and machine learning.

SciPy is a library that contains a collection of scientific and numerical algorithms, including optimization, linear algebra, integration, interpolation, and more. It is built on top of NumPy and provides additional functionality for scientific computing.

For example, you can use the minimize function from the scipy.optimize module to minimize a function. For example, you can use the following code to minimize the function f(x) = x^2 using the BFGS algorithm:

```
from scipy.optimize import minimize

def f(x):
    return x**2

res = minimize(f, x0=2, method='BFGS')
print(res.x) #The minimum point
```

Curve fitting is a process of finding the best-fit curve that describes the relationship between independent and dependent variables. SciPy provides the curve_fit function in the scipy.optimize module that can be used to fit a curve to a set of data points. The function takes a model function, initial parameters, and data as input, and returns the optimized parameters that best fit the data.
Here is an example of using the curve_fit function to fit a sine wave to a set of data points:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def sine_wave(x, a, b, c, d):
    return a * np.sin(b * x + c) + d

x_data = np.linspace(0, 10, 50)
y_data = sine_wave(x_data, 2, 0.5, 1, 0) + np.random.normal(0, 0.2, x_data.shape)

params, cov = curve_fit(sine_wave, x_data, y_data)
print(params)

plt.plot(x_data, y_data, 'o', label='data')
```

```
plt.plot(x_data, sine_wave(x_data, params[0], params[1], params[2], params[3]), '-', label='fit')
plt.legend()
plt.show()
```

SciPy also provides a range of metrics and optimization functions in the scipy.optimize module. For example, the minimize function can be used to minimize a given function with respect to its inputs, and the root_scalar function can be used to find the roots of a given function. The metrics module provides a range of performance metrics such as mean squared error, root mean squared error, mean absolute error, and R^2 score that can be used to evaluate the performance of a model.

Example of using curve_fit from scipy.optimize to fit a curve to a set of data points:

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Generate data to fit
x_data = np.linspace(0, 10, 50)
y_data = 2 * x_data + 3 + np.random.normal(0, 1, x_data.shape)

# Define the curve to fit
def linear_function(x, a, b):
    return a * x + b

# Fit the curve to the data
params, cov = curve_fit(linear_function, x_data, y_data)
a, b = params

# Plot the data and the fit
plt.scatter(x_data, y_data)
plt.plot(x_data, linear_function(x_data, a, b), color='red')
plt.show()
```

Regarding various metrics, the scipy library provides a number of functions for evaluating metrics like mean squared error, mean absolute error, R^2 score, and more. Here's an example using the mean squared error metric:

```
import numpy as np
from scipy.optimize import curve_fit
from sklearn.metrics import mean_squared_error

# Generate data to fit
```

```
x_data = np.linspace(0, 10, 50)
y_data = 2 * x_data + 3 + np.random.normal(0, 1, x_data.shape)

# Define the curve to fit
def linear_function(x, a, b):
    return a * x + b

# Fit the curve to the data
params, cov = curve_fit(linear_function, x_data, y_data)
a, b = params

# Calculate the mean squared error
y_pred = linear_function(x_data, a, b)
mse = mean_squared_error(y_data, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
```

Scikit-learn is a machine learning library that provides a wide range of supervised and unsupervised learning algorithms. It is built on top of NumPy and SciPy and is designed to be easy to use and efficient.

For example, you can use the following code to train a linear regression model on a dataset and make predictions:

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
import numpy as np

# Generating example data
np.random.seed(0)
X = np.random.rand(100, 1)
y = 2 + 3 * X + np.random.rand(100, 1)

# Splitting the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Training the model
reg = LinearRegression().fit(X_train, y_train)

# Making predictions
y_pred = reg.predict(X_test)
```

These libraries offer many powerful tools for optimization, linear algebra and machine learning, and are widely used in scientific computing, data analysis and machine learning projects.

Optimization is the process of finding the best solution to a problem within a set of constraints. In mathematical terms, it involves finding the values of a set of variables that minimize or maximize a given function, called the objective function.

Optimization is widely used in many fields such as finance, engineering, and operations research. There are many different optimization techniques, such as linear programming, nonlinear programming, integer programming, and stochastic optimization, that can be used to solve different types of optimization problems.

In Python, there are several libraries, such as scipy.optimize, that provide a wide range of optimization algorithms, such as gradient descent, Nelder-Mead, conjugate gradient, and BFGS. These libraries can be used to optimize various types of functions, including linear and nonlinear functions, with or without constraints.

For example, you can use the minimize function from the scipy.optimize module to minimize a function. For example, you can use the following code to minimize the function f(x) = x^2 using the BFGS algorithm:

```
from scipy.optimize import minimize

def f(x):
    return x**2

res = minimize(f, x0=2, method='BFGS')
print(res.x) #The minimum point
```

It is also possible to optimize functions with constraints using the minimize function, by providing a dictionary of constraints or using the method 'SLSQP'

```
from scipy.optimize import minimize

def f(x):
    return x[0]**2 + x[1]**2

def constraint1(x):
    return x[0] + x[1] - 1

def constraint2(x):
    return 1 - x[0] - x[1]

cons = [{'type': 'eq', 'fun': constraint1},
```

```
        {'type': 'ineq', 'fun': constraint2}]

res = minimize(f, x0=[0.5, 0.5], constraints=cons)
print(res.x) #The minimum point
```

In some cases optimization problems are not convex, and can have multiple local minimas. In these cases it is important to use optimization algorithms that are capable of finding global optima, like genetic algorithms, or simulated annealing, which are not provided in scipy.optimize.

Linear algebra is the branch of mathematics that deals with vector spaces and linear transformations between them. It is a fundamental tool in many areas of mathematics and science, including physics, engineering, computer science, and economics.

In Python, the most popular library for linear algebra is NumPy, which provides a wide range of functions and classes for working with arrays and matrices. It is also the foundation for other important libraries such as scikit-learn and pandas.

NumPy provides functions for basic linear algebra operations such as matrix multiplication, inversion, and determinant calculation, as well as more advanced operations such as eigenvalue and singular value decomposition.

For example, you can use the dot function to perform matrix multiplication:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.dot(A, B)
print(C)
```

The output will be:

```
[[19 22]
 [43 50]]
```

You can also use the inv function to calculate the inverse of a matrix:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
A_inv = np.linalg.inv(A)
print(A_inv)
```

The output will be:
[[-2. 1.]
 [1.5 -0.5]]

Furthermore, numpy.linalg module also provides functions for solving linear systems of equations, eigenvalue and eigenvector calculation, and singular value decomposition.

Linear algebra is a fundamental concept in machine learning and data science, it is used in many algorithms such as Principal Component Analysis (PCA), Singular Value Decomposition (SVD), Linear Discriminant Analysis (LDA), and many others.

Scikit-learn a machine learning library built on top of NumPy, makes use of NumPy's linear algebra capabilities, and provides a wide range of machine learning models that can be used for tasks such as classification, regression, and clustering.

Machine learning is a subfield of artificial intelligence (AI) that involves developing algorithms and statistical models that allow computers to learn from and make predictions or decisions without being explicitly programmed to do so.

There are several types of machine learning:

- Supervised learning: The algorithm is trained on a labeled dataset, where the correct output is provided for each input. The goal is to make predictions on new, unseen data. Examples include linear regression, logistic regression, and support vector machines.
- Unsupervised learning: The algorithm is trained on an unlabeled dataset, and the goal is to discover patterns or structure in the data. Examples include k-means clustering and principal component analysis (PCA)
- Reinforcement learning: The algorithm learns by interacting with an environment, receiving rewards or penalties for certain actions. The goal is to learn to make decisions that maximize the cumulative reward over time.
- Deep learning: A subfield of machine learning that involves training deep neural networks (DNN) on large amounts of data. DNNs have been shown to achieve state-of-the-art performance on a wide range of tasks such as image classification, natural language processing, and speech recognition.

For example, you can use scikit-learn to train a linear regression model on a dataset:

```
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

# Load the Boston housing dataset
boston = load_boston()
```

```
X = boston.data
y = boston.target

# Create and fit the model
model = LinearRegression()
model.fit(X, y)

# Make predictions
predictions = model.predict(X)
```

In this example, the LinearRegression class is used to create a linear regression model, the fit method is used to train the model on the Boston housing dataset, and the predict method is used to make predictions on new data.

Machine learning can be applied to a wide range of problems, such as image classification, natural language processing, speech recognition, predictive analytics, and many others.

Natural Language Processing (NLP) is a subfield of AI that involves developing algorithms and models for understanding, analyzing, and generating human language.

Machine learning is widely used in NLP for tasks such as:

- Text classification: classifying text into predefined categories, such as spam detection, sentiment analysis, and topic classification.
- Named entity recognition (NER): identifying and extracting entities such as people, organizations, and locations from text.
- Part-of-speech tagging (POS): labeling words in a sentence with their grammatical roles, such as nouns, verbs, and adjectives.
- Language modeling: predicting the next word in a sentence given the previous words.
- Machine Translation: translating text from one language to another.
- Text summarization: automatically summarizing the main points of a text.
- Sentiment Analysis: determining the sentiment of a given text.

For example, you can use scikit-learn library to train a text classification model on a dataset:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

#define dataset
X_train = ["This is a positive text", "This is a negative text", "This is a neutral text"]
y_train = ["positive", "negative", "neutral"]
X_test = ["This is a positive text", "This is a negative text"]
```

```
#vectorize text
vectorizer = CountVectorizer()
X_train_vectors = vectorizer.fit_transform(X_train)
X_test_vectors = vectorizer.transform(X_test)

#train model
clf = MultinomialNB()
clf.fit(X_train_vectors, y_train)

#predict
predicted = clf.predict(X_test_vectors)
print("Accuracy: ", accuracy_score(predicted, y_test))
```

In this example, the dataset is a list of texts and their corresponding labels ("positive", "negative", "neutral"). The CountVectorizer class is used to convert the texts into numerical feature vectors, the MultinomialNB class is used to train a Naive Bayes model on the dataset, and the predict method is used to make predictions on new data.

NLP is widely used in various industries such as customer service, chatbot, and virtual assistant, e-commerce, and social media.

Another example where you can use scikit-learn library to train a Named Entity Recognition (NER) model on a dataset:

```
import nltk
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction import DictVectorizer
from sklearn.tree import DecisionTreeClassifier

#define dataset
sentences = [["I", "live", "in", "New York"], ["I", "work", "at", "Google"]]
pos_tags = [["PRP", "VBP", "IN", "NNP"], ["PRP", "VB", "IN", "NNP"]]
ner_tags = [["O", "O", "O", "B-GPE"], ["O", "O", "O", "B-ORG"]]

#flatten the data
sentences = [token for sent in sentences for token in sent]
pos_tags = [tag for sent in pos_tags for tag in sent]
ner_tags = [tag for sent in ner_tags for tag in sent]

#extract features
```

```
def extract_features(sentence, index):
    features = {
        'word': sentence[index],
        'is_first': index == 0,
        'is_last': index == len(sentence) - 1,
        'is_capitalized': sentence[index][0].upper() == sentence[index][0],
        'is_all_caps': sentence[index].upper() == sentence[index],
        'is_all_lower': sentence[index].lower() == sentence[index],
        'prefix-1': sentence[index][0],
        'prefix-2': sentence[index][:2],
        'prefix-3': sentence[index][:3],
        'suffix-1': sentence[index][-1],
        'suffix-2': sentence[index][-2:],
        'suffix-3': sentence[index][-3:],
        'prev_word': '' if index == 0 else sentence[index - 1],
        'next_word': '' if index == len(sentence) - 1 else sentence[index + 1],
        'has_hyphen': '-' in sentence[index],
        'is_numeric': sentence[index].isdigit(),
        'capitals_inside': sentence[index][1:].lower() != sentence[index][1:]
    }
    return features

#extract features for each token
X = [extract_features(sentences, i) for i in range(len(sentences))]

#Vectorize the features
vec = DictVectorizer()
X = vec.fit_transform(X)

#split dataset into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, ner_tags, test_size=0.33)

#train the model
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

#predict
predicted = clf.predict(X_test)
print("Accuracy: ", accuracy_score(predicted, y
```

In addition to text classification, another common task in NLP that can be approached with machine learning is named entity recognition (NER). This involves identifying and classifying named entities, such as people, organizations, and locations, in text. One approach

to NER is to use a sequence labeling algorithm, such as a hidden Markov model (HMM) or a conditional random field (CRF), to predict the labels for each word in a sentence. Another approach is to use a machine learning model, such as a support vector machine (SVM) or a neural network, to classify words or phrases in the text.

In practice, many NLP tasks, such as part-of-speech tagging and syntactic parsing, can also be approached with machine learning. Another example of NLP task that can be approached with machine learning is sentiment analysis. Sentiment analysis is the process of determining whether a piece of text expresses a positive, negative, or neutral sentiment. Sentiment analysis is used in a wide range of applications, such as customer service, marketing, and opinion mining. Common techniques for sentiment analysis include using supervised machine learning algorithms to classify text based on the presence of certain keywords or using unsupervised techniques to identify patterns in the text.

Lecture 5

Image processing, OpenCV

Image processing is a technique of manipulating and analyzing digital images, and it is often used in computer vision and computer graphics. OpenCV (Open Source Computer Vision

Library) is a popular open-source library for image processing and computer vision in Python. It provides a wide range of tools and algorithms for image processing, such as image filtering, thresholding, color space conversion, feature detection, and more.

One of the most common tasks in image processing is image thresholding, which is the process of converting an image into a binary image (black and white) based on a threshold value. OpenCV provides several thresholding methods, such as binary thresholding, adaptive thresholding, and Otsu's thresholding.

Another common task in image processing is image filtering, which is the process of modifying the pixels of an image based on a filter kernel. OpenCV provides several image filtering methods, such as Gaussian blur, median blur, and bilateral filter.

Another common task in image processing is feature detection, which is the process of detecting and extracting features in an image, such as corners, edges, and circles. OpenCV provides several feature detection methods, such as Harris corner detection, FAST corner detection, and Hough Circle Transform

A Practical task with OpenCV could be to use the library to detect faces in an image. This can be done using a pre-trained classifier, such as the Viola-Jones algorithm, which is included in OpenCV. Another task could be to use OpenCV to detect and track objects in a video stream, which can be done using a variety of algorithms such as background subtraction, optical flow, and object tracking.

Overall, OpenCV is a powerful library that provides a wide range of tools for image processing and computer vision, making it a great choice for a wide range of applications such as image enhancement, object detection, and video analysis.

A few examples of using OpenCV for image processing tasks:

Thresholding: The following code demonstrates how to use OpenCV to perform binary thresholding on an image. The cv2.threshold() function is used to convert the image to a binary image based on a threshold value.

```
import cv2

# Load the image
img = cv2.imread("image.jpg")

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Perform binary thresholding
```

```
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

# Show the original and thresholded images
cv2.imshow("Original", img)
cv2.imshow("Thresholded", thresh)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Filtering: The following code demonstrates how to use OpenCV to perform median filtering on an image. The cv2.medianBlur() function is used to apply a median filter to the image.

```
import cv2

# Load the image
img = cv2.imread("image.jpg")

# Perform median filtering
median = cv2.medianBlur(img, 5)

# Show the original and filtered images
cv2.imshow("Original", img)
cv2.imshow("Median Filtered", median)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Feature Detection The following code demonstrates how to use OpenCV to detect and draw circles in an image. The cv2.HoughCircles() function is used to detect circles in the image using the Hough Circle Transform.

```
import cv2

# Load the image
img = cv2.imread("image.jpg")

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Detect circles in the image
circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1, 20, param1=50, param2=30,
minRadius=0, maxRadius=0)

# Draw the circles on the image
for circle in circles[0, :]:
```

```
    cv2.circle(img, (circle[0], circle[1]), circle[2], (0, 255, 0), 2)

# Show the original and processed images
cv2.imshow("Original", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

These examples are just a few examples of the image processing tasks that can be performed using OpenCV, and it is a powerful library for image processing and computer vision.

Binarization is the process of converting a continuous-tone image into a binary image, where the pixels in the image are either black or white. Binarization is typically used to segment an image into foreground and background, or to highlight specific features in an image.

Binarization is particularly useful in applications where images are used for character recognition, such as in optical character recognition (OCR) and handwriting recognition. For example, in OCR, the goal is to accurately recognize text in images of scanned documents, and binarization is used to separate the text from the background. Similarly, in handwriting recognition, binarization is used to separate the written characters from the paper.

Car number plate recognition is another application that uses binarization. In this application, the goal is to accurately recognize the characters on a car number plate. Binarization is used to separate the characters from the background, making it easier for the recognition algorithm to identify the characters.

There are various methods for binarization, including global thresholding, adaptive thresholding, and Otsu's method. Global thresholding sets a single threshold value for the entire image, and pixels above the threshold are set to white and pixels below the threshold are set to black. Adaptive thresholding sets a threshold value for each individual pixel, based on the values of surrounding pixels. Otsu's method automatically determines the optimal threshold value, based on the histogram of the image.

Binarization is a crucial step in various image analysis applications, including character recognition, handwriting recognition, and car number plate recognition. The choice of binarization method depends on the specific requirements of the application and the characteristics of the image being processed.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, filters

image = io.imread("text.png", as_gray=True)
threshold_value = filters.threshold_otsu(image)
binary_image = image > threshold_value
```

```
plt.imshow(binary_image, cmap='gray')
plt.show()
```

In this example, image is read in as a grayscale image using io.imread. The optimal threshold value for converting the grayscale image to binary is determined using the threshold_otsu function from the filters module. The resulting binary image can be visualized using imshow from matplotlib.

Here's another example that demonstrates the use of different binarization methods for recognizing handwritten text and car numbers:

```
import cv2
import numpy as np

# Load the image
image = cv2.imread("handwritten_text.jpg", cv2.IMREAD_GRAYSCALE)

# Apply Gaussian thresholding
gaussian = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)

# Apply Otsu's thresholding
_, otsu = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

# Show the results
cv2.imshow("Gaussian", gaussian)
cv2.imshow("Otsu", otsu)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In this example, the image is first loaded using cv2.imread and converted to grayscale using the cv2.IMREAD_GRAYSCALE flag. The cv2.adaptiveThreshold function is then used to apply Gaussian thresholding to the image. The cv2.threshold function is used to apply Otsu's thresholding to the image. The resulting binary images are displayed using cv2.imshow and can be compared to determine which method is more effective for recognizing the text or car numbers in the image.

Dilation and erosion are morphological image processing operations that are used to expand or shrink the boundaries of objects in an image. Dilation increases the size of the object by adding pixels to the object's boundary, while erosion reduces the size of the object by removing pixels from the object's boundary. These operations are often used in image analysis tasks such as object detection and image segmentation.

```
import cv2

# Load image
img = cv2.imread('image.jpg')

# Create a 3x3 square kernel for dilation
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))

# Apply dilation
dilated = cv2.dilate(img, kernel)

# Create a 3x3 square kernel for erosion
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))

# Apply erosion
eroded = cv2.erode(img, kernel)

# Save the dilated and eroded images
cv2.imwrite('dilated_image.jpg', dilated)
cv2.imwrite('eroded_image.jpg', eroded)
```

Methods for finding contours in an image involve identifying the boundaries of objects in the image. These methods are often used in image analysis tasks such as object detection and image segmentation. Examples of contour finding methods include the Canny edge detector, the Sobel operator, and the Laplacian of Gaussian operator.

Detection and segmentation of objects in an image involve identifying and isolating specific objects or regions of interest within an image. These tasks are often used in image analysis applications such as object recognition, object tracking, and image compression. Examples of object detection and segmentation methods include the Viola-Jones object detection framework, the R-CNN object detection framework, and the Mask R-CNN object detection and segmentation framework.

Examples of using OpenCV for image processing tasks include:

Finding contours:

```
import cv2

# Load image
```

```
img = cv2.imread('image.jpg')

# Convert image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply Canny edge detection
edges = cv2.Canny(gray, 50, 150)

# Find contours in the image
contours, hierarchy = cv2.findContours(edges, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

# Draw the contours on the original image
cv2.drawContours(img, contours, -1, (0, 255, 0), 3)

# Save the image with contours
cv2.imwrite('contoured_image.jpg', img)
```

Object detection and segmentation:

```
import cv2

# Load image
img = cv2.imread('image.jpg')

# Load a pre-trained object detection model
model = cv2.dnn.readNetFromCaffe('model.prototxt', 'model.caffemodel')

# Prepare the image for object detection
blob = cv2.dnn.blobFromImage(img, 1.0, (300, 300), (104.0, 177.0, 123.0))

# Feed the image through the model
model.setInput(blob)
detections = model.forward()

# Loop over the detections
for i in range(0, detections.shape[2]):
    confidence = detections[0, 0, i, 2]

    if confidence > 0.5:
        # Get
```

You can also use the inRange method to threshold the image and segment the object of a specific color.

```
import cv2

# Load the image
img = cv2.imread("image.jpg")

# Define the range of the color you want to segment
lower = (0, 0, 0)
upper = (50, 50, 50)

# Threshold the image to get only the desired color
mask = cv2.inRange(img, lower, upper)

# Show the segmented image
cv2.imshow("Segmented Image", mask)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Lecture 6

Web scraping is the process of automating the extraction of information from websites, often large amounts of data, in order to transform it into a structured format for analysis or storage. It is typically done using a specialized software tool or a general-purpose programming language such as Python, which has several libraries specifically designed for web scraping, such as Beautiful Soup and Scrapy. The goal of web scraping is to extract data from websites, typically by simulating a human browsing the web, and to retrieve specific information that can be used for various purposes such as data analysis, machine learning, and more.

The difference between scraping and parsing is that parsing typically refers to the process of breaking down a larger piece of data into smaller, more manageable parts. In the context of web scraping, parsing refers to the process of extracting specific information from the HTML response received from a website. Scraping, on the other hand, encompasses the entire process of sending the HTTP request, receiving the HTML response, and then parsing the response to extract the desired information.

Parsing LXML sites is the process of extracting data from a website using the LXML library in Python. LXML is a powerful library for processing XML and HTML documents, making it a popular choice for web scraping.

In this lecture, we will learn about the basics of LXML and how to use it to extract information from websites. We will start by discussing the basic structure of an XML or HTML document and how to parse it using LXML.

Next, we will learn about the different functions and methods available in LXML for selecting and extracting elements from a website. We will also discuss the different ways to handle the data once it has been extracted, such as transforming it into a more useful format or storing it in a database.

Finally, we will learn how to use LXML in combination with other Python libraries, such as Requests, to build more complex web scraping projects.

By the end of this lecture, you will have a good understanding of how to use LXML to parse websites and extract information from them. With this knowledge, you will be able to build your own web scraping projects and automate tasks that would otherwise require manual effort.

LXML is a Python library that provides an API for processing and manipulating XML and HTML data. It is widely used for web scraping and data extraction because of its ease of use, flexibility, and speed. LXML allows you to parse HTML and XML documents, extract information from them, and even manipulate the documents if necessary.

To get started with LXML, you'll need to install it by running pip install lxml. Once installed, you can import the library into your Python script by running import lxml.

In order to parse a website using LXML, you'll first need to retrieve the HTML or XML data from the site. You can do this using the requests library in Python, which provides an easy way to make HTTP requests. For example, the following code will retrieve the HTML from a website and store it in a variable:

```
import requests

response = requests.get("https://example.com")
html = response.text
```

Once you have the HTML data, you can parse it using LXML's fromstring method. This method takes a string of HTML or XML data and returns an object that you can manipulate. For example:

```
from lxml import html

root = html.fromstring(html)
```

The resulting root object represents the root node of the HTML or XML document. You can then use the methods provided by LXML to extract information from the document. For example, you can use the xpath method to search for elements with specific tag names or attributes:

```
headings = root.xpath("//h1")
```

This code will return a list of all <h1> elements in the HTML document. You can then iterate over the list and extract the text content of each heading:

```
for heading in headings:
    print(heading.text)
```

LXML also provides methods for manipulating the document, such as adding or removing elements, changing attribute values, or adding text content. You can use these methods to modify the HTML or XML data and then save it to a file or send it back to a server.

In this lecture, we have covered the basics of LXML and how to use it to extract information from websites. There are many more features and techniques available in LXML, so be sure to consult the official documentation for more information: https://lxml.de/

LXML is a library that provides an easy way to manipulate and extract information from XML and HTML documents. It is built on top of the libxml2 and libxslt C libraries, and provides a Pythonic API for working with these documents.

One of the key features of LXML is its ability to manipulate the structure of an XML or HTML document. This includes adding or removing elements, changing attribute values, or adding text content. For example, you can use the ElementTree API to navigate the document tree, access elements and attributes, and manipulate their values.

In order to extract information from a website using LXML, you first need to parse the HTML or XML content of the website into a tree-like structure. You can do this using the lxml.html or lxml.etree module, depending on whether you are working with HTML or XML content.

Once you have parsed the content into an ElementTree structure, you can use a variety of methods to extract the information you are interested in. For example, you can use the xpath method to select elements based on their tag names, classes, or other attributes, or you can use the cssselect method to select elements based on their CSS selectors.

In addition to these basic parsing and extraction capabilities, LXML also provides a range of additional features and tools, including support for parsing and validating documents against DTDs or XML schemas, support for transforming XML documents using XSLT stylesheets, and support for working with HTML forms and submitting data to websites.

Overall, LXML is a powerful and versatile library for working with XML and HTML documents, and is well-suited for a wide range of applications, including web scraping, data analysis, and document processing.

ElementTree is a part of the python standard library and is used for working with XML and HTML files. It provides a simple way to parse and manipulate XML/HTML files.

The main class in ElementTree is the Element class, which represents an XML element. It has methods for accessing and modifying the element's tag, attributes, text, and children elements.

Here's a basic example of how to use ElementTree to parse an XML file and access its elements:

```
import xml.etree.ElementTree as ET

# parse an XML file
tree = ET.parse('sample.xml')
root = tree.getroot()

# access elements
for child in root:
    print(child.tag, child.attrib)
    for subchild in child:
        print(subchild.tag, subchild.attrib)

# modify elements
for elem in root.iter('value'):
    elem.text = 'new value'

# write the modified XML to a file
tree.write('modified.xml')
```

This example shows how to parse an XML file using ElementTree, access its elements, modify their values, and write the modified XML back to a file.

lxml is a library for working with XML and HTML documents in Python. It provides two main modules: lxml.etree and lxml.html.

lxml.etree is used for working with XML documents and provides a robust and feature-rich API for parsing, manipulating, and generating XML and HTML. It is based on the ElementTree API, which is included in the Python standard library, but adds many additional features and optimizations.

lxml.html is used for working with HTML documents and provides a convenient API for parsing and manipulating HTML, as well as cleaning up the HTML to make it well-formed and ready for processing.

Both lxml.etree and lxml.html are fast and memory-efficient, making them suitable for working with large documents or scraping large numbers of web pages. They also

support a wide range of encoding types, including Unicode and various ISO-8859 encodings, making them suitable for working with documents in multiple languages.

Once you have parsed the content into an ElementTree structure, you can use a variety of methods to extract the information you are interested in. For example, you can use the xpath method to select elements based on their tag names, classes, or other attributes, or you can use the cssselect method to select elements based on their CSS selectors.

In addition to these basic parsing and extraction capabilities, LXML also provides a range of additional features and tools, including support for parsing and validating documents against DTDs or XML schemas, support for transforming XML documents using XSLT stylesheets, and support for working with HTML forms and submitting data to websites.

An example of parsing an HTML document with LXML and extracting information:

```
import lxml.html

# Load the HTML document into memory
html = lxml.html.fromstring("<html><body><h1>Hello World!</h1></body></html>")

# Use the xpath method to select the h1 element
h1 = html.xpath("//h1")[0]

# Extract the text content of the h1 element
print(h1.text)  # "Hello World!"
```

An example of parsing an XML document with LXML and transforming it using an XSLT stylesheet:

```
import lxml.etree

# Load the XML document into memory
xml = lxml.etree.fromstring("<data><record><name>John Doe</name><age>30</age></record></data>")

# Load the XSLT stylesheet into memory
```

```
xslt = lxml.etree.fromstring("""
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <body>
        <table>
          <tr>
            <th>Name</th>
            <th>Age</th>
          </tr>
          <xsl:for-each select="data/record">
            <tr>
              <td><xsl:value-of select="name" /></td>
              <td><xsl:value-of select="age" /></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
""")

# Transform the XML document using the XSLT stylesheet
result = lxml.etree.XSLT(xslt)(xml)

# Print the transformed HTML document
print(lxml.etree.tostring(result, pretty_print=True).decode())
```

These are just a few examples of what you can do with LXML. For more information on the capabilities of LXML and how to use it to work with XML and HTML documents, you can refer to the official documentation: https://lxml.de/index.html

BeautifulSoup is another popular library for web scraping and parsing HTML and XML documents. Like LXML, it provides a way to extract information from websites, but it has a slightly different focus and approach. While LXML is a more comprehensive library that provides a wide range of functionality for manipulating and transforming XML and HTML documents, BeautifulSoup is focused more

specifically on the task of web scraping, and provides a simpler and more straightforward interface for extracting information from websites.

BeautifulSoup is particularly well-suited for tasks such as extracting text content from HTML pages, extracting data from tables, or finding specific elements on a page based on their tag names, classes, or attributes. To use BeautifulSoup, you first need to install the library and import it into your Python code, and then use the BeautifulSoup function to parse the HTML or XML document you want to scrape.

Once you have parsed the document, you can use a variety of methods to search for and extract the information you are interested in. For example, you can use the find and find_all methods to search for specific elements on the page, or you can use the select method to search for elements that match a particular CSS selector.

Overall, BeautifulSoup is a powerful and flexible library that is widely used by web developers and data scientists for tasks such as web scraping, data analysis, and text processing. Whether you are working with HTML, XML, or other types of documents, BeautifulSoup provides a simple and intuitive way to extract the information you need and turn it into actionable insights and data.

One of the key features of BeautifulSoup is its ability to search for tags and elements using tag names, class and id attributes, and CSS selectors. For example, you can use the "select" method to select elements based on their CSS selectors, or the "find" method to find the first element matching a given tag name or attribute.

Another important feature of BeautifulSoup is its ability to handle malformed HTML code. Unlike other libraries that may fail when parsing malformed HTML, BeautifulSoup is able to parse and extract data from even the most poorly formatted HTML documents.

Here is an example of using BeautifulSoup to extract data from an HTML document:

```
from bs4 import BeautifulSoup

html_doc = """
<html>
  <head>
    <title>Example Page</title>
  </head>
```

```
    <body>
        <h1>Welcome to my page</h1>
        <p>Here is some content</p>
        <p class="highlight">And here is a highlighted paragraph</p>
    </body>
</html>
"""

soup = BeautifulSoup(html_doc, "html.parser")

# Extract the title tag
title = soup.title

# Extract the first h1 tag
h1 = soup.h1

# Extract all p tags
p_tags = soup.find_all("p")

# Extract the first p tag with class "highlight"
highlighted_p = soup.select_one(".highlight")

print("Title:", title.text)
print("H1:", h1.text)
print("P tags:")
for p in p_tags:
    print(p.text)
print("Highlighted P:", highlighted_p.text)
```

This code will output the following:

```
Title: Example Page
H1: Welcome to my page
P tags:
Here is some content
And here is a highlighted paragraph
Highlighted P: And here is a highlighted paragraph
```

Once you have extracted the information from a website using either LXML or BeautifulSoup, the next step is to process the text information you have received. This

can involve a range of tasks, such as cleaning and pre-processing the data, transforming it into a format suitable for analysis, or extracting specific information from the text, such as named entities, keywords, or sentiment.

One common example of processing text information from a website is the extraction of named entities from a news article. For example, consider the following article: *"Microsoft has announced that it is acquiring LinkedIn for $26.2 billion. The deal is expected to close by the end of 2016"*.

To extract named entities from this article, you would first need to clean and pre-process the text, such as converting all characters to lowercase and removing punctuation. You would then use a named entity recognition (NER) tool, such as the Named Entity Recognizer (NER) in the Natural Language Toolkit (NLTK), to identify the named entities in the text.

Here is an example of how this could be done in Python using NLTK:

```
import nltk
from nltk.tokenize import word_tokenize

text = "Microsoft has announced that it is acquiring LinkedIn for $26.2 billion. The deal is expected to close by the end of 2016."

# Tokenize the text
tokens = word_tokenize(text)

# Part-of-speech tag the tokens
tagged = nltk.pos_tag(tokens)

# Identify named entities
named_entities = nltk.ne_chunk(tagged)

# Print the named entities
for entity in named_entities:
    if hasattr(entity, 'label'):
        print(entity.label(), ' '.join(c[0] for c in entity.leaves()))
```

This code will produce the following output:

```
ORGANIZATION Microsoft
```

ORGANIZATION LinkedIn
MONEY $26.2 billion

As you can see, the NER tool has correctly identified the named entities "Microsoft" and "LinkedIn" as organizations, and the amount "$26.2 billion" as money.

Lecture 7

Application architecture refers to the design of an application's underlying systems, including the structure of its components, the relationships between these components, and the principles that guide their design. A well-designed application architecture can help ensure that an application is scalable, maintainable, and easy to develop and deploy.

API, or Application Programming Interface, is a set of protocols, routines, and tools for building software applications. It specifies how software components should interact and APIs allow for communication between different applications or systems. APIs can also be used to access remote systems or data, making it easier for developers to integrate their applications with other services.

A common use of APIs is in the creation of microservices, where an application is broken down into smaller, reusable components that can be managed and developed independently. APIs can also be used to expose data and functionality to external users, allowing them to build their own applications that interact with the system.

For example, a weather service might provide an API that allows developers to access its data and use it in their own applications. Another example is the use of APIs in social media platforms, which allows developers to access information about users, posts, and other data, and use it to build their own applications.

Application architecture and APIs play a crucial role in modern software development and are essential for creating scalable, flexible, and maintainable applications.

The application could have a front-end interface that allows users to enter their location and view the current weather conditions. The back-end of the application would then use an API to retrieve weather data from a weather service provider, such as OpenWeatherMap.

The application architecture in this example would include the following components:

- Front-end interface: This is the user-facing part of the application that allows users to enter their location and view the weather information.
- API: The application uses an API to retrieve weather data from the weather service provider. This could be a REST API or a SOAP API, depending on the weather service provider's specifications.
- Weather service provider: This is the source of the weather data that the application retrieves through the API.

In this example, the front-end interface sends a request to the weather service provider's API, asking for the current weather conditions for a specific location. The API then returns the weather data in a format that the application can process, such as JSON or XML. The front-end interface then displays the weather information to the user.

Example of a simple API written in Python using the Flask library:

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/api', methods=['GET', 'POST'])
def handle_api_request():
    if request.method == 'GET':
        # handle GET request
        return 'Handling GET request'
    elif request.method == 'POST':
        # handle POST request
        return 'Handling POST request'

if __name__ == '__main__':
    app.run()
```

This example sets up a basic API with a single endpoint that handles both GET and POST requests. When a GET request is received, the API will return the string "Handling GET request", and when a POST request is received, it will return the string "Handling POST request".

In a real-world API, you would likely perform some more complex operations, such as querying a database or performing some other action based on the contents of the request. But this example should give you a basic understanding of how an API works and how you can create one using Python and Flask.